

EDF 调度实验报告

杜天蛟 141250031

2016-11-26

基本 EDF 调度实现

思路与方法

- 在 TCB 中添加关于 EDF 实现的信息

```
typedef struct edf_data {
    INT32U c_value;    // the task needs to consume c_value ticks in
    p_value ticks as a period, const
    INT32U p_value;    // const
    INT32U comp_time;  // the task remain to consume in a period
    INT32U ddl;        // the deadline of a task
    INT32U start;
    INT32U end;
}EDF_DATA;
```

其中 `c_value` 和 `p_value` 是常量，任务一旦创建就不会改变，`comp_time` 是一个周期内还剩多少 ticks 没有执行，`ddl` 是当前该任务的 deadline，这两个值会根据时钟和调度而更新。`start` 和 `end` 是为了计算 `comp_time` 和 `ddl` 方便添加的变量。

TCB 的结构体中有个名为 `OSTCBExtPtr` 的 `void*` 指针，可以指向用户自定义的 TCB extension。因此创建任务时只需要把初始化过的 `edf_data` 这个结构体传入就可以了。

- 创建自己的任务函数：

```
static void task1(void *pdata) {
    OSTimeDly(1);
    while (1) {
        while (((EDF_DATA*)OSTCBCur->OSTCBExtPtr)->comp_time > 0) {
            //do nothing
        }
    }
}
```

仿照 `app.c` 中的 `AppTaskStart()` 的创建方式创建任务。定义任务栈，定义优先级和 ID，并把自己定义好的 `edf_data` 传入。

```
OSTaskCreateExt((void*)(void *))task1,
    (void *)0,
    (OS_STK *)&TASK1STK[TASK_STK_SIZE - 1],
    (INT8U)TASK_1_PRIO,
    (INT16U)TASK_1_ID,
    (OS_STK *)&TASK1STK[0],
    (INT32U)TASK_STK_SIZE,
    (void *)&edf_datas[0],
    (INT16U)0);
```

为了让任务同时进入系统，在每个任务开始时都加上了 `OSTimeDly(1)`

- 编写一个新的任务调度函数 OS_SchedEDF，在每个时钟周期内遍历任务列表，找到 ddl 最近的任务，并把该任务的优先级赋值给 OSPrioHighRdy。这样系统就可以选择优先级为 OSPrioHighRdy 的任务执行了。

```
static void OS_SchedEDF(void) {
    OS_TCB* p_current;
    OS_TCB* edf_ptcb;
    int temp_earliest_deadline = 1000000;
    int temp_deadline = 0;
    int isAllDelay = 1;

    p_current = OSTCBLIST;
    edf_ptcb = OSTCBLIST;
    OS_ENTER_CRITICAL();
    while (p_current->OSTCBPrio != OS_TASK_IDLE_PRIO) {
        //if the task is ready and not delay

        if (p_current->OSTCBPrio < 60 && p_current->OSTCBDly == 0 &&
            ((EDF_DATA*)p_current->OSTCBExtPtr)->comp_time > 0) {
            temp_deadline = ((EDF_DATA*)p_current->OSTCBExtPtr)->ddl;
            if (temp_deadline < temp_earliest_deadline) {
                temp_earliest_deadline = temp_deadline;
                edf_ptcb = p_current;
            }
            isAllDelay = 0;
        }
        p_current = p_current->OSTCBNext;
    }
    OSPrioHighRdy = edf_ptcb->OSTCBPrio;
    if (isAllDelay == 1) {
        OSPrioHighRdy = OS_TASK_IDLE_PRIO;
    }
    OS_EXIT_CRITICAL();
}
```

- 在 OSTimeTick 中修改当前任务的 comp_time，更新该任务的 ddl, start, end 等值。

```
OS_ENTER_CRITICAL();
((EDF_DATA*)OSTCBCur->OSTCBExtPtr)->comp_time--;
if (((EDF_DATA*)OSTCBCur->OSTCBExtPtr)->comp_time == 0) {
    ((EDF_DATA*)OSTCBCur->OSTCBExtPtr)->ddl =
    ((EDF_DATA*)OSTCBCur->OSTCBExtPtr)->ddl +
    ((EDF_DATA*)OSTCBCur->OSTCBExtPtr)->p_value;
    ((EDF_DATA*)OSTCBCur->OSTCBExtPtr)->comp_time =
    ((EDF_DATA*)OSTCBCur->OSTCBExtPtr)->c_value;
    ((EDF_DATA*)OSTCBCur->OSTCBExtPtr)->end = OSTimeGet();
}
```

```

OSTCBCur->OSTCBDly =
    (((EDF_DATA*)OSTCBCur->OSTCBExtPtr)->p_value -
    (((EDF_DATA*)OSTCBCur->OSTCBExtPtr)->end -
    ((EDF_DATA*)OSTCBCur->OSTCBExtPtr)->start);
    ((EDF_DATA*)OSTCBCur->OSTCBExtPtr)->start =
    ((EDF_DATA*)OSTCBCur->OSTCBExtPtr)->start +
    ((EDF_DATA*)OSTCBCur->OSTCBExtPtr)->p_value;
    printf("\n%d\tComplete\t%d\t%d",
OSTimeGet(),OSTCBCur->OSTCBId,getEDFNextID());
    }
    else {
        if (OSTCBCur->OSTCBId != getEDFNextID()) {
            APP_TRACE("\n%d\tPreempt\t\t%d\t%d",
OSTimeGet(),OSTCBCur->OSTCBId,getEDFNextID());
        }
    }
    OS_EXIT_CRITICAL();

```

算法很简单，如果该任务的 `comp_time` 为 0，说明该任务已经完成，那就顺次后移 `ddl` 和 `start`，重置 `comp_time`，否则不做改变。如果该任务的 `comp_time` 不为 0，则说明被抢占。为了将任务切换过程打印出来，又添加了 `getEDFNextID()` 的方法，其中算法和 `OS_SchedEDF()` 中一样，找到下一个 `ddl` 最近的任务的 `Id` 并返回。

- 因为包的问题，有时候无法用 UC/OS2 定义好的 `APP_TRACE` 的宏，所以我又在 `core.c` 上面 `#include<stdio.h>`，使用 `printf` 来打印
- 为了方便查看输出结果，将 `os_cfg.h` 中的 `OS_TICKS_PER_SEC` 修改为 1u

测试

- TaskSet l = { t1(1,3), t2(3,6) }

输出：

```
D:\temp\Micrium_Win32_OS2\Micrium\Software\Evalboards\Microsoft\Windows\OS2\Visual Studio\Debug\OS2.exe
OSTick created, Thread ID 1416
Task[ 63] created, Thread ID 4828
Task[ 62] created, Thread ID 11200
Task[ 61] created, Thread ID 9712
Task[ 21] created, Thread ID 8744
Task[ 22] created, Thread ID 3188
Task[ 21] '?' Running
Task[ 22] '?' Running
Task[ 63] 'uC/OS-II Idle' Running

1 Preempt 65535 1
2 Complete 1 2
5 Complete 2 1
6 Complete 1 65535
7 Preempt 65535 1
8 Complete 1 2
11 Complete 2 1
12 Complete 1 65535
13 Preempt 65535 1
14 Complete 1 2
17 Complete 2 1
18 Complete 1 65535
19 Preempt 65535 1
20 Complete 1 2
```

• TaskSet 2 = { $t_1(1,3)$, $t_2(3,6)$, $t_3(4,9)$ }

输出：

```
选择D:\temp\Micrium_Win32_OS2\Micrium\Software\Evalboards\Microsoft\Windows\OS2\Visual Studio\Debug\OS2.exe
OSTick created, Thread ID 6852
Task[ 63] created, Thread ID 10892
Task[ 62] created, Thread ID 11800
Task[ 61] created, Thread ID 5008
Task[ 21] created, Thread ID 1824
Task[ 22] created, Thread ID 5216
Task[ 23] created, Thread ID 332
Task[ 21] '?' Running
Task[ 22] '?' Running
Task[ 23] '?' Running
Task[ 63] 'uC/OS-II Idle' Running

1 Preempt 65535 1
2 Complete 1 2
5 Complete 2 1
6 Complete 1 3
10 Complete 3 1
11 Complete 1 2
14 Complete 2 3
18 Complete 3 65535
19 Preempt 65535 3
23 Complete 3 65535
28 Preempt 65535 3
32 Complete 3 65535
37 Preempt 65535 3
41 Complete 3 65535
```

EDF 算法改进

算法描述

EDF 算法决定优先级的机制过于单一，仅仅依靠每个任务请求的绝对截止时间来，并不考虑任务是否重要，不能很好的反映实际要求。可能会出现在任务截止时间相同的情况下，一

个软实时要求的任务得到调度，而另一个硬实时要求的任务得不到调度。

因此可以引入一个表示任务重要性的 importance 变量，用这个 importance 变量和任务的绝对截止时间来共同决定任务调度时的优先级。这样，当两个任务的 ddl 相同时，有硬实时要求的任务就会比有软实时要求的任务先得到调度。

改进后的 EDF 算法具体描述如下：

- 每个任务到来时需要提供周期、执行时间、优先级、Id 和表示重要程度的 level 信息，其中 level 越小，意味着越重要。为每个 level 准备一个队列，用于存放本级别的就绪任务，任务在队列中按照绝对截止时间由小到大排列。对于截止时间相同的任务，level 低的任务应在 level 高的任务前调度。
建立一个数组，数组的每个元素指向其下标所对应的任务 level 队列，可称之为 level 队列数组。就绪任务到来时先根据 level 找到对应的 level 队列，再根据截止时间加入该队列中。每个队列的队首任务就是本队列中绝对截止时间最小的任务。在每个时钟周期内，系统搜索所有 level 队列的队首任务，选取截止时间最小的任务，并更新该队列。
- 如图所示，

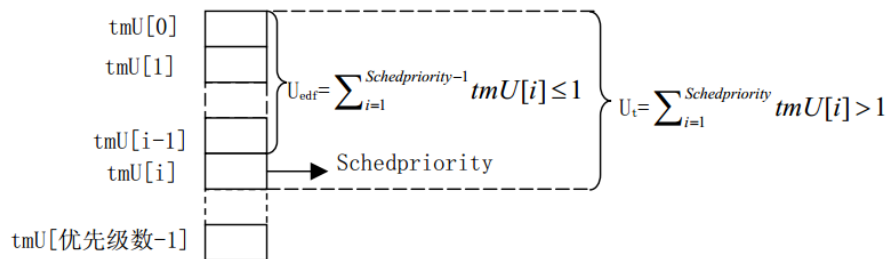


图 3-8 $tmU[优先级数]$ 、 U_{edf} 、 $Schedpriority$ 和 U_t 关系

在系统中定义变量 $tmU[level]$ 、 U_{edf} 、 $Schedpriority$ 和 U_t 用于标示系统任务中 EDF 算法可以调度的子任务集。 $tmU[level]$ 用于记录每个 level 的所有任务 CPU 利用率之和。在系统轻载情况下， U_t 是系统中所有 level 的就绪任务的 CPU 利用率之和，即 $U_t = \sum_{i=1}^{优先级数} tmU[i]$ ， n 为系统中所有任务的个数。系统过载情况下， U_t 、 U_{edf} 和 $Schedpriority$ 满足的关系如图 3-8 所示。其中 $U_{edf} = \sum_{i=1}^{Schedpriority-1} tmU[i] \leq 1$ 并且 $U_{edf} = \sum_{i=1}^{Schedpriority-1} tmU[i] > 1$ 。 $Schedpriority$ 表示 EDF 算法可调度子任务集的 level 界限。level 小于 $Schedpriority$ 的任务可参与系统调度，level 大于 $Schedpriority$ 的任务不可参与系统调度。level 等于 $Schedpriority$ 的任务根据该任务的 CPU 使用率 u 与 $U - U_{edf}$ 的关系决定，当 $u \leq U - U_{edf}$ 时可以参与系统调度；当 $u > U - U_{edf}$ 时不参系统调度。当有新的任务到达或者需要从任务队列中删除任务时，系统需要重新计算 U_{edf} 和 $Schedpriority$ 。具体方法是：按 level 从低到高累加每个 level 的所有任务的 CPU 使用率，当累加值大于 1 时停止累加。 $Schedpriority$ 等于此时的 level。 U_{edf} 等于累加值减去 $tmU[Schedpriority]$ 。计算 U_{edf} 和 $Schedpriority$ 实际上是在对任务集进行可调度性判定。在综合考虑系统开销和判定效果后，本算法使用公式 $U = \sum_{i=1}^n C_i/T_i \leq 1$ (执行时间为 C_i ，周期为 T_i) 作为可调度性的判定依据。此外随着时间的流逝，一个任务的某次请求在可调度性判定之前没有得到调度，由于其剩余时间减小，

此请求的紧迫程度增大，此刻的时限要求类似于周期为其剩余时间的周期任务（算法假设任务的截止时间等于周期）。所以在进行可调度性判定时使用此任务请求的剩余时间代替周期，即 $u' = C/(d-t)$ (t 为系统当前时间)。

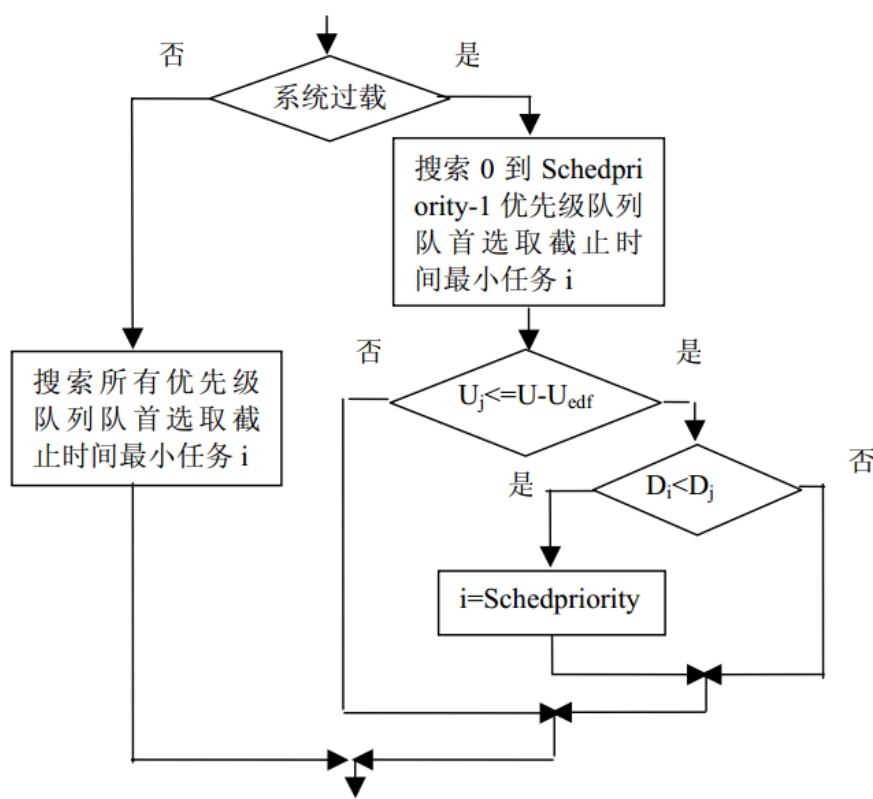


图 3-9 改进后 EDF 算法

- 对已经丢弃的任务，在整个实时系统恢复轻载状态时，其原有的 deadline 重新设置为当前时间 $t + ddl$ ，重新进入调度。

改进算法实现

关键数据结构

- ```
typedef struct queue_data {
 INT32U prio;
 INT32U deadline;
 INT32U importance;
 INT32U id;
}Queue_Data;
```

Queue\_Data 和每一个 TCB 对应，里面的 prio, deadline, id 和 TCB 中的对应属性始终保持一致。Importance 属性表示该任务所属的等级

- ```
typedef struct myqueue {
    INT32U capacity;
    INT32U size;
    Queue_Data *elements;
```

```

}Queue;
Queue taskQueue[QUEUE_LEVEL];

```

定义一个任务等级队列数组，每一个等级对应一个队列，elements 指向该队列的队首。

```

void initQueue();
void enqueue(Queue_Data* data);
void update(INT32U level);

```

在系统初始化时调用 initQueue()，为队列分配内存。

在创建任务时调用 enqueue()，初始化该任务对应的 Queue_Data 后，将其放在对应的队伍中去。
(按 ddl 升序排)

在 OSTimeTick()方法中，当该任务在该周期已经执行完毕，那么更新 TCB 属性后更新该任务所在队列的排序。

核心算法

```

static void OS_SchedEDFQueue() {
    int i;
    int earliest_deadline = 1000000;
    int earliest_prio = 0;
    int earliest_level = 0;
    OS_ENTER_CRITICAL();
    for (i = 0; i < QUEUE_LEVEL; i++) {
        Queue_Data* first = taskQueue[i].elements;
        if (first->deadline < earliest_deadline) {
            earliest_deadline = first->deadline;
            earliest_prio = first->prio;
            earliest_level = i;
        }
    }
    OS_TCB* p_current = OSTCBLIST;
    int isAllDelay = 1;
    while (p_current->OSTCBPrio != OS_TASK_IDLE_PRIO) {
        //if the task is ready and not delay
        if (p_current->OSTCBPrio < 60 && p_current->OSTCBDly == 0 &&
            ((EDF_DATA*)p_current->OSTCBExtPtr)->comp_time > 0) {
            isAllDelay = 0;
        }
        p_current = p_current->OSTCBNext;
    }
    OSPrioHighRdy = earliest_prio;
    level = earliest_level;
    if (isAllDelay == 1) {
        OSPrioHighRdy = OS_TASK_IDLE_PRIO;
    }
}

```



```
OS_EXIT_CRITICAL();
```

```
}
```

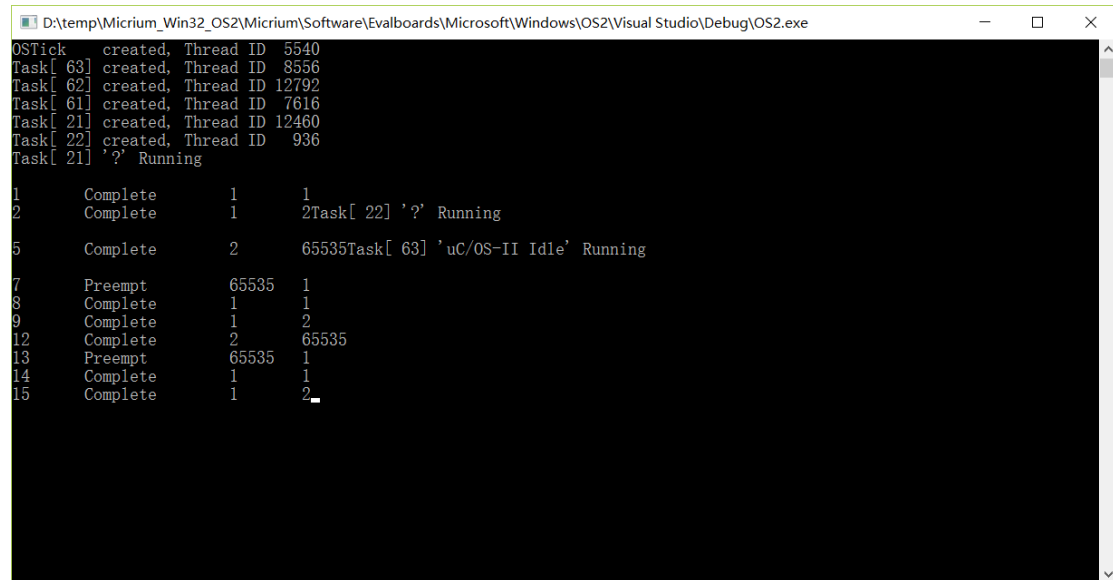
即遍历每个队列的队首，找到 `ddl` 最小的一个任务执行。

测试

输入：

```
• TaskSet I = { t1(1,3), t2(3,6) }
```

输出：



```
D:\temp\Micrium_Win32_OS2\Micrium\Software\Evalboards\Microsoft\Windows\OS2\Visual Studio\Debug\OS2.exe
OSTick created, Thread ID 5540
Task[ 63] created, Thread ID 8556
Task[ 62] created, Thread ID 12792
Task[ 61] created, Thread ID 7616
Task[ 21] created, Thread ID 12460
Task[ 22] created, Thread ID 936
Task[ 21] '?' Running

1 Complete 1 1
2 Complete 1 2Task[ 22] '?' Running
5 Complete 2 65535Task[ 63] 'uC/OS-II Idle' Running
7 Preempt 65535 1
8 Complete 1 1
9 Complete 1 2
12 Complete 2 65535
13 Preempt 65535 1
14 Complete 1 1
15 Complete 1 2
```